intel®

Intel386™ Family

Program Development Templates

Order Number 481894-001

PROGRAM DEVELOPMENT TEMPLATES

Order Number: 481894-001

REV.	REVISION HISTORY	DATE
-001	Original Issue.	01/89

In the United States, additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ®.)

Above	iLBX	Intellink	MICROMAINFRAME	Ripplemode
BITBUS	im®	iOSP	MULTIBUS®	RMX/80
COMMputer	iMDDX	iPAT	MULTICHANNEL	RUPI
CREDIT	iMMX	iPDS	MULTIMODULE	Seamless
Data Pipeline	Inboard	iPSC	ONCE	SLD
ETOX	Insite	iRMK	OpenNET	SugarCube
FASTPATH	Intel®	iRMX®	OTP	UPI
Genius	intel®	iSBC®	PC BUBBLE	VLSiCE
Δ i	Intel376	iSBX	Plug-A-Bubble	376
i®	Intel386	iSDM	PROMPT	386
I^2ICE	intelBOS	iSXM	Promware	386SX
ICE	Intel Certified	KEPROM	QueX	387
iCEL	Intelevision	Library Manager	QUEST	387SX
iCS	Inteligent Identifier	MAPNET	Quick-Erase	4-SITE
iDBP	Inteligent Programming	MCS®	Quick-Pulse Programming	
iDIS	Intellec®	Megachassis		

VAX and VMS are trademarks of Digital Equipment Corporation.

Contents

Chapter 1 Creating the Example Template Systems

Chapter 2 Exploring the Templates

Appendix

Glossary

Index

Figures

Tables

Chapter 1
Creating the Example Template Systems

Creating a flat (linear) model bootloadable system for an Intel386™ family microprocessor is straightforward. BLD386, the system builder, assigns absolute addresses and makes the necessary system data structures. Several working examples are included on your diskette. You can build the example systems and monitor their execution on an in-circuit emulator without prior knowledge of the Intel386 architecture, and without knowing how the systems work; just follow the the step-by-step directions in Section 1.2.

Chapter 2 contains discussions of the templates and the flat memory models they represent. Certain Intel-specific terms used here may be unfamiliar to you, such as 32-bit protected mode, descriptor, segment, or gate. If you come across unfamiliar terms, or if you are interested in learning more about the Intel386 architecture, see the Glossary at the end of this booklet or the *Introduction to the 80386* or the *80386 Programmer's Reference Manual* listed at the end of this chapter.

1.1 The Example Software

The software files are templates because they are patterns you can build upon and modify in creating your own bootloadable system. There is code for three different C applications and an assembler application, each of which can be built into a bootloadable system. Three different kinds of systems can be built:

- a simple RAM system that runs without any interrupt handlers
- a typical embedded system with code in ROM and data and tables in RAM
- a system protected from stack overflow

The templates include all of the source code, some sample applications, the commands necessary to build the sample systems, and commands for an ICE-386™ or ICE-376™ in-circuit emulator to demonstrate the sample systems. The tables that follow group the files by the type of example system they build.

FILES FOR SIMPLE EXAMPLE	
File name	Description
simpinit.asm	microprocessor initialization code
cstart.asm	startup code for C application
simple.c	simple C-386 application
simple.bld	BLD386 system definitions
simple.bat (DOS) **simple.com** (VMS)	commands for assembling, compiling, binding, and building
showsimp.inc	**simple** system emulator commands

FILES FOR EMBEDDED EXAMPLE WITH INTERRUPT STUBS	
File name	Description
flatinit.asm	microprocessor initialization code
flatintr.asm	interrupt stubs
cstart.asm	startup code for C application
bitcount.c	C-386 application
reverse.c	C-386 application
flat.bld	BLD386 system definitions
flat.bat (DOS) **flat.com** (VMS)	commands for assembling, compiling, binding, and building
showbitc.inc	**bitcount** system emulator commands
showrevr.inc	**reverse** system emulator commands

FILES FOR PROTECTED EXAMPLE	
File name	Description
protinit.asm	microprocessor initialization code
protintr.asm	interrupt procedure for asmstart
asmstart.asm	startup code for ASM386 application
protect.asm	ASM386 application
protect.bld	BLD386 system definitions
protect.bat (DOS) **protect.com** (VMS)	commands for assembling, binding, and building
showprot.inc	**protect** system emulator commands

1.2 Executing the Examples

You can execute the examples by following these simple directions.

1) Please ensure that your development system has the following software and hardware:

REQUIRED HARDWARE AND SOFTWARE		
Name	Version	Description
asm386	V3.Ø	Macro Assembler
c386	V1.Ø	C-386 C Compiler
bnd386	RLLv1.3	Binder (V1.3)
bld386	RLLv1.3	System Builder (V1.4)
ice386 ice376		ICE-386 or ICE-376 In-Circuit Emulator on DOS host

2) Copy all the template files to your working directory. For VMS, the template files are located in the directory

SYS&ROOT:[SYSHLP.EASE]. For DOS, the files are located on the diskette entitled **Program Development Templates**. All files are ASCII text.

3) To create an example system, invoke the appropriate file of commands.

DOS	simple system:	**simple**
	embedded system:	**flat bitcount** or **flat reverse**
	protected system:	**protect**
VMS	simple system:	**@simple**
	embedded system:	**@flat bitcount** or **@flat reverse**
	protected flat system:	**@protect**

These commands assemble, compile (if necessary), bind, and build the system. The assembler and builder issue valid warning messages designed to guide careful creation of the systems. You can ignore these messages.

4) For VMS, download your system (**simple, bitcount, reverse,** or **protect**) and corresponding file of emulator commands (**showsimp.inc, showbitc.inc, showrevr.inc,** or **showprot.inc**) to your DOS in-circuit emulator host.

5) On your DOS in-circuit emulator host, change to your ICE-386 or ICE-376 in-circuit emulator software directory. Copy the system (**simple, bitcount, reverse,** or **protect**) from your working directory to your in-circuit emulator software directory. Then copy the corresponding file of emulator commands (**showsimp.inc, showbitc.inc, showrevr.inc,** or **showprot.inc**) from your working directory to your in-circuit emulator software directory.

6) Press the reset button on the in-circuit emulator. Invoke the in-circuit emulator.

ice386 or
ice376

7) At the -> prompt, include the sequence of commands written for you by typing in one of the following followed by a carriage return.

```
->include nolist showsimp.inc   /* for showing simple   */
->include nolist showbitc.inc   /* for showing bitcount */
->include nolist showrevr.inc   /* for showing reverse  */
->include nolist showprot.inc   /* for showing protect  */
```

8) To start the commands, at the -> prompt type the name of the system you have created followed by a carriage return.

```
->simple                                /* to view simple   */
->bitcount                              /* to view bitcount */
->reverse                               /* to view reverse  */
->protect                               /* to view protect  */
```

9) Follow the execution of your system, entering a carriage return when prompted.

The ICE-386 in-circuit emulator issues error #-26t on the **istep** command. Though the message is valid, you can ignore it because the emulator chip is executing in real mode and no selectors are accessible.

The sequence of emulator commands ends when the emulator returns you to the -> prompt. You can view another sample system that you have created by returning to step 7 of these directions, or you can exit the emulator. To exit the emulator, type **exit**.

```
->exit
```

1.3 Related Publications

The following publications contain further information on the Intel386 family of microprocessors.

Intel386™ Family System Builder User's Guide, order number 481342

Intel386™ Family Utilities User's Guide, order number 481343

Introduction to the 80386, order number 231252

80386 Programmer's Reference Manual, order number 230985

80386 System Software Writer's Guide, order number 231499

ASM386 Assembly Language Reference Manual, order number 480251

ASM386 Macro Assembler Operating Instructions for DOS Systems, order number 451290

ASM386 Macro Assembler Operating Instructions for VAX/VMS Systems, order number 167675

C-386 User's Guide, order number 481378

ICE™-376 In-Circuit Emulator User's Guide, order number 481753

ICE™-386 In-Circuit Emulator User's Guide, order number 481404

ICE™-386/25 In-Circuit Emulator User's Guide, order number 481749

ICE™-386SX™ In-Circuit Emulator User's Guide, order number 451989

1.4 Trademarks

Intel386, 376, 386, 386SX, and ICE are trademarks of Intel Corporation.

VAX and VMS are trademarks of Digital Equipment Corporation.

Chapter 2
Exploring the Templates

You can use the templates in many different systems. They are easy to understand and are at the heart of most embedded, 32-bit protected-mode systems.

Section 2.1 describes the flat model for 32-bit protected-mode programming. It explores the **flat** control for the system builder and the build file of system definitions. Section 2.2 describes the templates for the embedded examples with interrupt stubs. Most of this information applies to the other example systems as well. Section 2.3 describes adding protection to the flat model. Section 2.4 describes the protected example templates, including implementing a protected expand-down stack.

2.1 Protected-Mode Programming

Your code must initialize the Intel386™ family microprocessor to run in 32-bit protected mode after system reset. Each example system has code to initialize the microprocessor to run in protected mode. The key that simplifies the initialization is using the system builder, BLD386, which creates the absolute 32-bit protected-mode system from your code.

The builder builds an absolute 32-bit protected-mode system from the linked input segments you provide. You also provide a build file that defines system implementation details. This build file may contain information such as protection levels, system data structure definitions, and memory configuration. Each example system has a template build file.

The Intel386 family of microprocessors enables you to program in 32-bit protected mode using a flat memory model. The flat model is easy to use because all address space is linear. The flat model is good for systems that require large amounts of memory for data or code.

2.1.1 The Default Flat Memory Model

The feature of the system builder that makes the flat memory model easy to use is the **flat** control. Using the **flat** control, you can implement many details of a flat model system automatically. When **flat** is used, the builder:

- Creates two segments called **_phantom_code_** and **_phantom_data_**. The default descriptor privilege level (DPL) value is zero, or most privileged. These overlaid segments each have a default base address of zero and a default limit of 4 gigabytes - 1. Figure 2-1 depicts the default flat memory model used by the builder.
- Combines input segments into the appropriate phantom segment in the order of input. Even though the segments are overlaid, the builder does not overlap true code and data. When ROM areas and RAM areas are specified in the build file, the builder allocates space for segments according to their access attributes: ROM receives read-only segments (tables and task state segments) and executable segments (code), RAM receives read-write segments (data and stack).

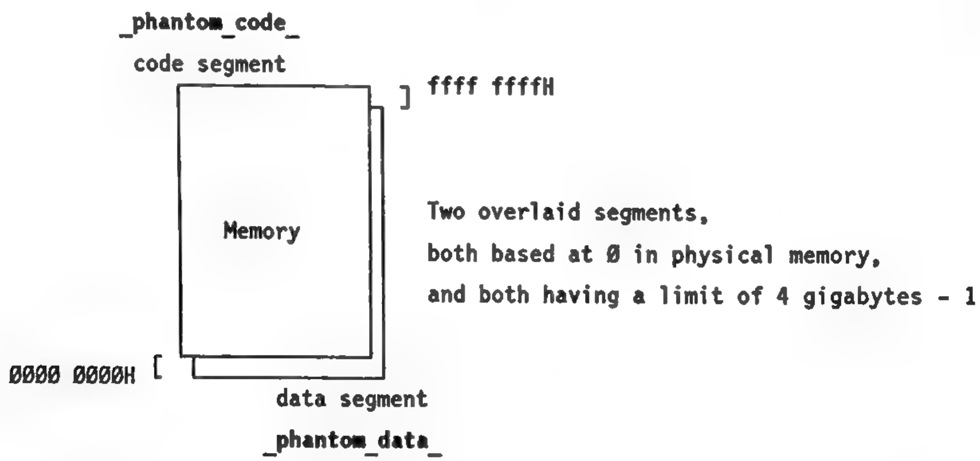

Figure 2-1 Default Flat Memory Model

- Adjusts gates and all relative and absolute jumps to use the base of the **_phantom_code_** descriptor.
- Adjusts all data and stack accesses to use the base of the **_phantom_data_** descriptor.

2.2 The Embedded Example Templates

The embedded example templates demonstrate the code necessary for a typical embedded application with interrupt stub routines. The initialization, startup, and interrupt code is written in ASM386 assembly language. The sample application is written in C-386. Figure 2-2 illustrates the structure of the files that contain the source code. Note that the compilation of the C application produces a **data** segment and a **code32** segment. The binder combines segments with like names. See Figure 2-3 in Section 2.2.6 to see how the modules become a bootloadable system.

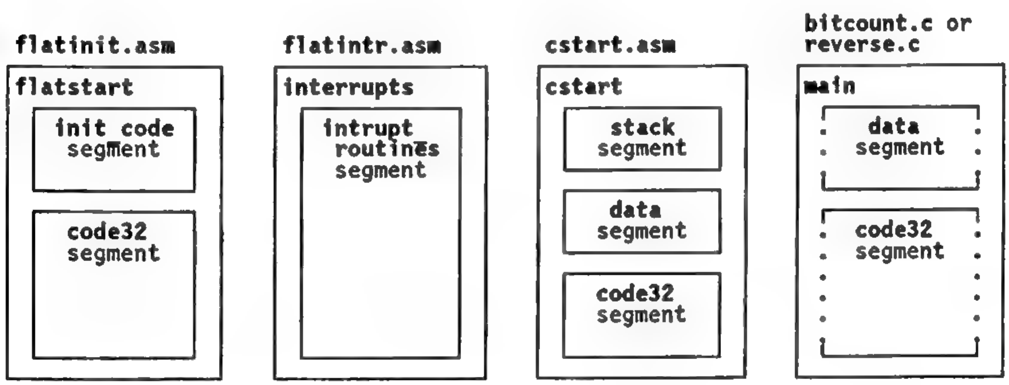

Figure 2-2 Embedded Example Source Code

2.2.1 The Initialization Template

The initialization code template, **flatinit.asm**, contains two distinct parts. The first part, the **init_code** segment, places the microprocessor into 32-bit protected mode. Since the code runs on both the 386™ microprocessor (which, at reset, is in 16-bit real mode)

and the 376™ microprocessor (which, at reset, is in 32-bit mode), the **init_code** segment is a 32-bit segment. For correct execution on a 386 microprocessor, operand prefixes before some instructions change the default operand size.

The other part of the initialization code, **copytables** in the **code32** segment, copies the necessary parts of the descriptor tables created by the builder from ROM to RAM. In this example, ROM is in high memory and RAM is in low memory. The RAM address of the descriptor tables are calculated by ANDing the corresponding ROM address with 0000ffffH. The code copies 5 global descriptor table (GDT) entries and the first 17 interrupt descriptor table (IDT) entries down into RAM. This example has only 16 interrupt routines, for interrupts 0 through 14 and 16.

NOTE FOR ROM-BASED SYSTEMS

If you are implementing a ROM-based system like this example, then the process of copying the descriptor tables to RAM may be important. The microprocessor attempts to write to the GDT descriptor's ACCESS bit upon execution of an **lgdt** or **lgdtw** instruction (load the global descriptor table register, GDTR). The GDT is initially in ROM. Unless the GDTR is loaded with a RAM-based address, your hardware must return a READY upon a write to ROM.

If your hardware does indeed return a READY after a write to ROM, then the step of copying descriptor tables down to RAM in the **copytables** routine is unnecessary. Ensure that your initialization code also loads the interrupt descriptor table register (IDTR) and simply replace the far jump destination at the end of the **code32** segment with that of your startup code (**c_startup** in the example).

After the code copies the necessary parts of each table from ROM to RAM, the code adjusts the base and limit values in the GDT and IDT alias descriptors in the global descriptor table.

A listing of the **flatinit.asm** file follows.

```
;  flatinit.asm
;  Initialization code for flat (linear) model example
;
;  ***************************************************************************
;
;  Version 2.0
;  Copyright Intel Corp., 1988
;  This template is intended for your benefit in developing applications/
;  systems using Intel386(TM) family microprocessors.  Intel hereby grants
;  you permission to modify and incorporate it as needed.
;
;  ***************************************************************************
;
;  This is initialization code to put the 386(TM) processor or 376(TM)
;  processor into flat mode.  It should work with all applications,
;  but this model makes certain assumptions.  The memory model is a typical
;  embedded application model:  descriptor tables and code reside
;  in ROM and data is in RAM.  This example assumes that ROM begins at
;  0ffff0000H; since descriptor tables may need to be RAM-based for
;  protected-mode execution, the code copies the builder-created GDT and IDT
;  down into RAM with the RAM address being <ROM_address AND 0000ffffH>.  It
;  assumes five GDT entries:  NULL, a GDT alias, an IDT alias, code, and data.
;  The builder creates the GDT alias and IDT alias and places them,
;  by default, in GDT[1] and GDT[2].  After entering protected mode,
;  this code jumps to an ASM386 startup routine for a C application.  You
;  can change this JMP address to that of your code, or make the label of
;  your code C_STARTUP.

NAME flatstart             ; name of object module

EXTRN c_startup:near       ; this is the label jumped to after init_code
                           ;  and copytables

pe_flag        equ 1       ; for setting PE bit
galias_off     equ 8       ; offset of GDT alias in GDT
ialias_off     equ 10H     ; offset of IDT alias in GDT
data_selc      equ 20H     ; offset of _phantom_data_ in GDT (GDT[4])
gdt_lim        equ 27H     ; assume that 5 entries are all that are needed in GDT
idt_lim        equ 87H     ; assume that 17 entries are all that are needed in IDT

CODEMACRO      opprefx     ; macro to change default operand size
               db 66H
ENDM

init_code      SEGMENT ER PUBLIC

;  GDT_DESC and IDT_DESC are public symbols referred to in the build file.
;  The LOCATION definitions in the TABLE section of the build file point to
;  these labels;  the builder stores the base and limit for the named table
;  at this location in memory.

PUBLIC         gdt_desc
PUBLIC         idt_desc

gdt_desc       dp  ?
idt_desc       dp  ?
```

```
; START is a label that points to the true beginning of our executable
; code.  The BOOTSTRAP control causes the builder to place a short jump to
; the named label (in this case, START) at the component reset vector.

PUBLIC        start

; Since this code initializes either a 386 or 376 processor into protected
; mode, the first instructions at START test for component type.
; The 386 processor at reset is in real or compatibility mode:  the PE bit is
; off and the D bit for CS is not set.  Instructions execute in their 16-bit
; form.  The 376 processor at reset has the PE bit on as well as the D bit,
; so instructions execute in their 32-bit form.

    nop                         ; NOPs are for initializing a 386
    nop                         ;  processor
start:
    cld                         ; clear direction flag
    smsw bx                     ; check for processor (376 or 386) at reset
    test bl,1                   ; use SMSW rather than MOV for speed
    jnz pestart

; Loading the GDTR at REALSTART or PESTART depends on user hardware
; returning a READY after a write to ROM.

realstart:                      ; is a 386 processor and in 16-bit real mode
    opprefx                     ; use operand prefix to
    mov eax,offset gdt_desc     ;  get 32-bit address of GDT pointer
    opprefx                     ; use operand prefix to
    and eax,0ffffh              ;  make address relative to reset area
    lgdtw cs:[eax]              ; load 24 bits of base into GDTR

    mov ax,bx                   ; copy machine status word
    or al,pe_flag               ; set PE bit
    lmsw ax                     ; load machine status word
    jmp next                    ; flush prefetch queue

pestart:                        ; is a 376 processor in 32-bit protected mode
    mov eax,offset gdt_desc     ; get 32-bit address of GDT pointer
    and eax,0ffffh              ; make address relative to reset area
    lgdt cs:[eax]               ; load 32 bits of base into GDTR

next:
    xor eax,eax                 ; initialize data selectors
    mov al,data_selc            ; GDT[4] is _phantom_data_
    mov ds,ax
    mov ss,ax
    mov es,ax
    mov fs,ax
    mov gs,ax
    test bl,1
    jnz pejump
```

```
;  Use C_STARTUP as the destination of this next jump if your hardware does
;  return a READY on a write to ROM, and skip the COPYTABLES routine.

    opprefx                  ; use operand prefix for 386 processor jump
pejump:
    jmp far ptr copytables    ; first far jump causes A31-A20 to drop low

init_code ENDS

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

code32  SEGMENT ER PUBLIC        ; BND386 combines this with other code32
                                 ;  segments

copytables:

;  Copy GDT and IDT from ROM to RAM.  Assume the RAM address for the tables is
;  the ROM address AND 0000ffffH.  This code uses the second GDT entry,
;  the GDT alias at GDT[1] = (base of GDT) + galias_off.

    mov eax,offset gdt_desc      ; get address of gdt_desc
    mov ebx,dword ptr [eax]+2    ; base of GDT is 2 bytes past gdt_desc

;  Move the GDT descriptors from ROM to RAM.

    mov esi,ebx                  ; source of move is present GDT base
    and ebx,0ffffH               ; calculate address of new GDT
    mov edi,ebx                  ; destination of move is calculated address
    mov ecx,gdt_lim+1            ; count of move is 5 entries X 8 bytes each
    rep movsb                    ; move 5 descriptors from ROM to RAM

;  Modify the GDT alias at GDT[1] to reflect the new base and limit of
;  the RAM-based GDT.  The GDT alias is a data segment descriptor
;  (read/write) which allows future modification of the GDT.
;  Reload the GDTR with the new base and limit values.  We do this by
;  changing the GDT alias to reflect the new base and limit, saving the
;  changes, setting up the GDT alias to reload the GDTR, reloading, and then
;  restoring the GDT alias.  EBX holds the new base address of the GDT.

                                 ; change base in second dword of GDT alias
    and dword ptr [ebx]+galias_off+4,0ffff00H
                                 ; change limit in GDT alias
    mov word ptr [ebx]+galias_off,gdt_lim
                                 ; save part of GDT alias
    mov edx,dword ptr [ebx]+galias_off+2
                                 ; set up new base for loading GDTR
    mov dword ptr [ebx]+galias_off+2,ebx
                                 ; reload the GDTR
    lgdt pword ptr [ebx]+galias_off
                                 ; restore saved part of GDT alias
    mov dword ptr [ebx]+galias_off+2,edx

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;  IDT mov from ROM to RAM

    mov eax,offset idt_desc      ; get address of idt_desc
    mov edx,dword ptr [eax]+2    ; base of IDT is 2 bytes past idt_desc
```

```
; Move the IDT descriptors from ROM to RAM.

    mov esi,edx              ; source of move is present IDT base
    and edx,0ffffH           ; calculate address of new IDT
    mov edi,edx              ; destination of move is calculated address
    mov ecx,idt_lim+1        ; count of move is 17 entries X 8 bytes each
    rep movsb                ; move 17 descriptors from ROM to RAM

; Modify the IDT alias descriptor at GDT[2] to reflect the new base and limit
; values for the RAM-based IDT and load the IDTR. EBX holds the address of
; the new GDT. EDX holds the address of the new IDT.

                             ; change base in second dword of IDT alias
    and dword ptr [ebx]+ialias_off+4,0ffff00H
                             ; change limit in IDT alias
    mov word ptr [ebx]+ialias_off,idt_lim
                             ; save part of IDT alias
    mov ecx,dword ptr [ebx]+ialias_off+2
                             ; set up new base for loading IDTR
    mov dword ptr [ebx]+ialias_off+2,edx
                             ; load the IDTR
    lidt pword ptr [ebx]+ialias_off
                             ; restore saved part of IDT alias
    mov dword ptr [ebx]+ialias_off+2,ecx

    jmp c_startup            ; jump to startup code

code32 ENDS

END
```

2.2.2 The Build File Template

The **flat.bld** build file containing system definitions is both simple and generically useful. The following briefly explains the use of each definition.

SEGMENT definition — sets the descriptor privilege levels (**DPLs**) of all input segments to zero, or most privileged. The builder creates the overlaid **_phantom_code_** and **_phantom_data_** segments when the **flat** control is used; they are included here as a reminder, even though their default descriptor privilege level is zero.

TABLE definition — (first occurrence) defines the global descriptor table (named **GDT**).

TABLE definition's **LOCATION** specification	places the absolute base address and limit (describing the GDT) into memory defined in the initialization code. The symbol **gdt_desc** is an uninitialized 6-byte area in the initialization module. The **LOCATION** feature is handy when you are relocating code often: since BLD386 always places the correct base and limit values in memory, you can change the location of either your initialization module (containing **gdt_desc**) or of your GDT and re-build without re-assembling any code.
TABLE definition's **BASE** specification	absolutely locates the GDT at the specified address in memory. In this example the GDT goes in ROM. Knowing the exact base address of this table is useful when debugging.
TASK definition	defines a task for the system. If the system data structures representing a task are in the bootloadable system, then the in-circuit emulator initializes the processor and makes ready to execute the task. A flat model system does not require a task, however. The demonstration emulator commands reset the emulator to execute the initialization code.
GATE definition	creates gate descriptors. The 386 and 376 microprocessors require that interrupt descriptor table entries be interrupt, task or trap gates. Instead of creating gate descriptors in assembly language, BLD386 creates them with the **GATE** definition. The **ENTRY** specification for each gate specifies the public label of the interrupt handlers.
TABLE definition	(second occurrence) defines the interrupt descriptor table (named **IDT**), sets the location to store the base and limit of the table to be **idt_desc** in **init_code**, and places the interrupt gates in the table.
MEMORY definition	describes the physical memory setup of the hardware, and defines the location of the software system.

MEMORY definition's **RESERVE** specification	specifies holes in the address space, or defines address ranges used in other ways (such as destinations of copied descriptor tables). BLD386 cannot place any code or data in this area of memory.
MEMORY definition's **RANGE** specification	determines which address ranges are RAM or ROM. When ROM and RAM areas are specified in the build file, the builder allocates space for segments according to their access attributes: ROM receives read-only segments (tables and task state segments) and executable segments (code); RAM receives read-write segments (data and stack).
TABLE definition	(last occurrence) tells the builder that the default local descriptor table (which we named **LDT1**) it creates should not be put in the bootloadable system. The contents of the table does appear in the builder listing, however.

A listing of the `flat.bld` build file follows.

```
--  flat.bld
--  Build file for input to BLD386 to create flat model example
--
--  **************************************************************************
--
--  Version 2.0
--  Copyright Intel Corp., 1988
--  This template is intended for your benefit in developing applications/
--  systems using Intel386(TM) family microprocessors.  Intel hereby
--  grants you permission to modify and incorporate it as needed.
--
--  **************************************************************************
--

flat; -- build program id

SEGMENT
    *segments       (DPL = 0),      -- Give all user segments a DPL of 0.
    _phantom_code_  (DPL = 0),      -- These two segments are created by
    _phantom_data_  (DPL = 0);      -- the builder when the FLAT control is used.
                                    -- Their default DPL is 0; they are
                                    -- listed here for reference only.

TABLE
            -- create GDT
    GDT                             -- GDT_DESC is a public symbol in
                                    -- the "flatstart" initialization module.
        (LOCATION = gdt_desc,       -- In the buffer starting at GDT_DESC,
                                    -- BLD386 places the GDT base and
                                    -- GDT limit values.  Buffer must be
                                    -- 6 bytes long.  The base and limit
                                    -- values are places in this buffer
                                    -- as two bytes of limit plus
                                    -- four bytes of base in the format
                                    -- required for use by LGDT instruction.
         BASE = 0ffff0100H
        );  -- end GDT

TASK                                -- Task is for ICE(TM)-386 or
    main_task                       -- ICE(TM)-376 emulator initialization.
        (BASE = 0ffff0200H,
         DATA = data,               -- Points to a segment that
                                    -- indicates initial DS value.
         CODE = main,               -- Entry point is main, which
                                    -- must be a public id.
         STACKS = (stack),          -- Segment id points to stack
                                    -- segment. Sets the initial SS:ESP.
         NO INTENABLED              -- Disable interrupts.
        );

GATE
    int0_gate   (INTERRUPT, DPL = 0, ENTRY = int0),
    int1_gate   (INTERRUPT, DPL = 0, ENTRY = int1),
    int2_gate   (INTERRUPT, DPL = 0, ENTRY = int2),
    int3_gate   (INTERRUPT, DPL = 0, ENTRY = int3),
    int4_gate   (INTERRUPT, DPL = 0, ENTRY = int4),
    int5_gate   (INTERRUPT, DPL = 0, ENTRY = int5),
    int6_gate   (INTERRUPT, DPL = 0, ENTRY = int6),
```

```
    int7_gate  (INTERRUPT, DPL = 0, ENTRY = int7),
    int8_gate  (INTERRUPT, DPL = 0, ENTRY = int8),
    int9_gate  (INTERRUPT, DPL = 0, ENTRY = int9),
    int10_gate (INTERRUPT, DPL = 0, ENTRY = int10),
    int11_gate (INTERRUPT, DPL = 0, ENTRY = int11),
    int12_gate (INTERRUPT, DPL = 0, ENTRY = int12),
    int13_gate (INTERRUPT, DPL = 0, ENTRY = int13),
    int14_gate (INTERRUPT, DPL = 0, ENTRY = int14),
    int16_gate (INTERRUPT, DPL = 0, ENTRY = int16);

TABLE
            -- create IDT
    IDT                              -- IDT_DESC is a public symbol in the
        (LOCATION = idt_desc,        -- "flatstart" initialization module.
                                     -- In the buffer starting at IDT_DESC
                                     -- the builder places two bytes of the IDT
                                     -- limit and four bytes of the IDT base
                                     -- values in the format required for use
                                     -- by LIDT instruction.
         BASE = 0ffff0000H,

                                     -- Slots 0 through 31 are Intel-reserved

         ENTRY = (0:int0_gate,   1:int1_gate,   2:int2_gate,   3:int3_gate,
                  4:int4_gate,   5:int5_gate,   6:int6_gate,   7:int7_gate,
                  8:int8_gate,   9:int9_gate,  10:int10_gate, 11:int11_gate,
                 12:int12_gate, 13:int13_gate, 14:int14_gate, 16:int16_gate)
        );  -- end IDT

 MEMORY
     (RESERVE = (0..250H),           -- For copying down IDT and GDT.
      RANGE = (          -- begin configuration section --
              rom_area = ROM(0ffff0000H..0fffffff0H),
              ram_area = RAM(251H..0ffffH)
             )           -- end configuration section   --
     );

 TABLE
     ldt1 (NOT CREATED);             -- Builder does not place LDT in object
                                     -- module, but contents appear in listing.

 END
```

2.2.3 The Interrupt Stubs Template

A listing of the interrupt stubs in the file **flatintr.asm** follows. Intel reserves the first 32 interrupts. Those which have been defined have a stub.

```
; flatintr.asm
; Interrupt fault handlers for use with flat model example
;
; ****************************************************************************
;
; Version 2.0
; Copyright Intel Corp., 1988
; This template is intended for your benefit in developing applications/
; systems using Intel386(TM) family microprocessors.  Intel hereby
; grants you permission to modify and incorporate it as needed.
;
; ****************************************************************************
;


NAME interrupts
intrupt_routines SEGMENT ER PUBLIC

PUBLIC int0,int1,int2,int3,int4,int5,int6,int7,int8,int9,int10,int11
PUBLIC int12,int13,int14,int16

; If an exception occurs, the corresponding fault handler is
; entered.  The interrupt number is pushed on top of the stack.

int0 :  push 00H            ; divide error
        jmp entrez
int1 :  push 01H            ; debug exceptions
        jmp entrez
int2 :  push 02H            ; non-maskable interrupt
        jmp entrez
int3 :  push 03H            ; breakpoint
        jmp entrez
int4 :  push 04H            ; overflow
        jmp entrez
int5 :  push 05H            ; bounds check
        jmp entrez
int6 :  push 06H            ; invalid opcode
        jmp entrez
int7 :  push 07H            ; coprocessor not available
        jmp entrez
int8 :  push 08H            ; double fault
        jmp entrez
int9 :  push 09H            ; coprocessor segment overrun
        jmp entrez
int10 : push 0aH            ; invalid tss
        jmp entrez
int11 : mov ax,ds           ; segment not present
        mov ds,ax           ; ensure full loading of the segment registers
        mov ax,es
        mov es,ax
        push 0bH
        jmp entrez
int12 : mov ax,ds           ; stack exception
        mov ds,ax           ; ensure full loading of the segment registers
        mov ax,es
        mov es,ax
        push 0cH
        jmp entrez
int13 : push 0dH            ; general protection
```

```
        jmp entrez
int14 : push 0eH            ; page fault
        jmp entrez
int16 : push 10H            ; coprocessor error

entrez : hlt

intrupt_routines ENDS

END
```

2.2.4 The C Startup Template

The code in **cstart.asm** defines a stack for a C application. The stack pointer is initialized, and the C routine is called. A listing of the **cstart.asm** file follows.

```
;  cstart.asm
;  An ASM386 module to initialize the stack and call a C application
;
;  ************************************************************************
;
;
;  Version 2.0
;  Copyright Intel Corp., 1988
;  This template is intended for your benefit in developing applications/
;  systems using Intel386(TM) family microprocessors. Intel hereby
;  grants you permission to modify and incorporate it as needed.
;
;  ************************************************************************
;
NAME cstart                 ; name of the object module
EXTRN main:near             ; label of the C application to be called
PUBLIC c_startup            ; public symbol used in processor initialization code

stack STACKSEG 1024

data SEGMENT RW PUBLIC
data ENDS

code32 SEGMENT ER PUBLIC

c_startup:
   mov esp,stackstart stack ; initialize stack pointer
   call main                ; call C application
   hlt                      ; halt processor

code32 ENDS

END
```

2.2.5 Creating the Embedded Example System

Your template software includes a file of commands for assembling, compiling, binding, and building your 32-bit protected-mode system.

The three invocations of ASM386 create object modules **flatinit.obj**, **flatintr.obj** and **cstart.obj**. The assembler issues warnings with each invocation due to the use of privileged instructions in the files. The **debug** control directs ASM386 to include extra information useful in symbolic debugging. The listing files are **flatinit.lst**, **flatintr.lst**, and **cstart.lst**.

VMS:

```
asm386/debug flatinit.asm
asm386/debug flatintr.asm
asm386/debug cstart.asm
```

DOS:

```
asm386 flatinit.asm debug
asm386 flatintr.asm debug
asm386 cstart.asm debug
```

The invocation of C-386 creates an object module ***parameter*.obj**, where the ***parameter*** is replaced with the name of the application code file without its extension. The **regallocate** control directs the compiler to optimize the allocation of register variables. The **code** control causes placement of a pseudo-assembly language listing at the end of the listing file. **Debug** directs C-386 to include extra information useful in symbolic debugging. The listing file is ***parameter*.lst**.

VMS:

```
c386/debug/regallocate/code 'p1.c
```

DOS:

```
c386 %1.c debug regallocate code
```

BND386 combines the input segments and resolves symbolic addressing. The **noload** control directs the binder to create a linkable (rather than loadable) file. The **debug** control indicates that the binder does not purge debug information. **Object** directs the output file to be named ***parameter*.bnd**. The listing file is ***parameter*.mpl**.

VMS:

```
bnd386/noload/debug/object='p1.bnd 'p1.obj, flat.obj, cstart.obj
```

DOS:

```
bnd386 %1.obj, flat.obj, cstart.obj noload debug object (%1.bnd)
```

The goal is an absolute bootloadable file (all addresses fixed in memory) suitable for loading into an ICE™-386 or ICE™-376 in-circuit

emulator. BLD386 creates such an absolute module, necessary descriptor tables, and a task for initializing the emulator. The **buildfile** control identifies **flat.bld** as the build file. The **bootstrap** control identifies the symbol **start** as the label of the instruction to be jumped to by the bootstrap jump placed at 0fffffff0H. The **flat** control directs the builder to configure the file in a flat model, where all code resides in the **_phantom_code_** segment and all data resides in the **_phantom_data_** segment. The **mod376** control causes the builder to issue messages to guide creation of the object module for a 376 processor. The warning about overlapping memory is such a guide. You can remove this control to create an object module for a 386 processor. The listing file is ***parameter*.mp2**. The final system is ***parameter***.

VMS:

```
bld386/buildfile=flat.bld/bootstrap=start/flat/mod376 'p1.bnd,intrpt.obj
```

DOS:

```
bld386 %1.bnd,intrpt.obj buildfile (flat.bld) bootstrap (start) flat mod376
```

2.2.6 The Embedded Example System

Figure 2-3 shows what the memory looks like with the location of code, data, and system data structures after building the system with a small C application.

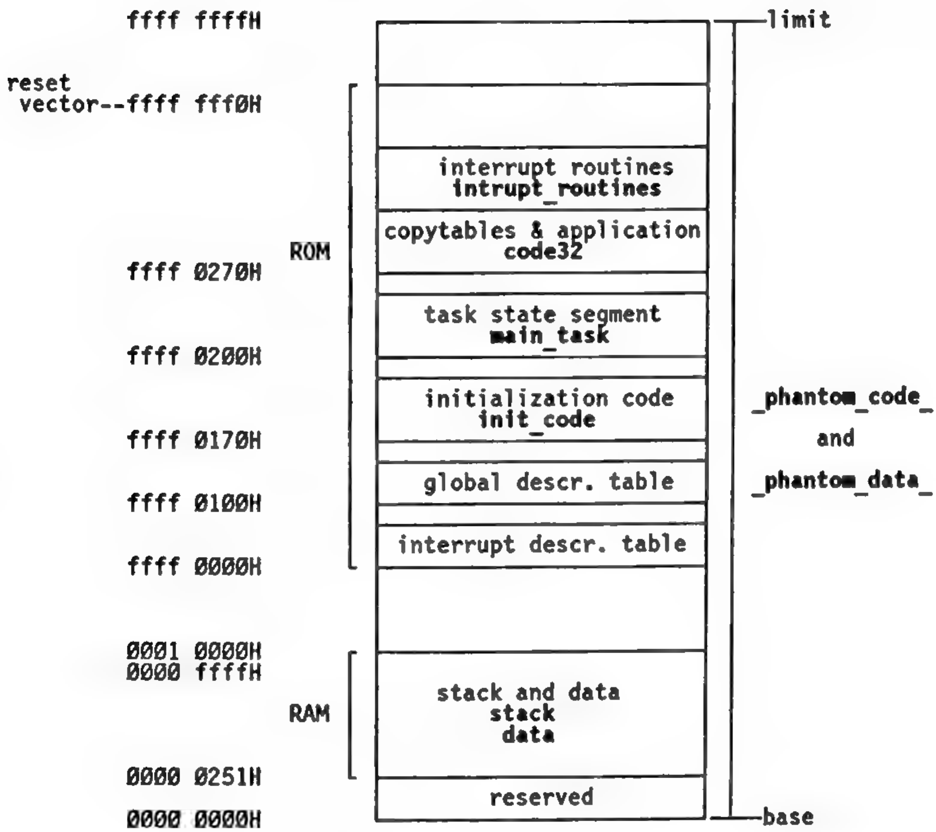

Figure 2-3 Embedded Example System Memory Map

The maps portion of the builder listing file, **bitcount.mp2**, for the **bitcount** example system follows.

80386 SYSTEM BUILDER
12/14/88 15:12:29   PAGE   3

SEGMENT MAP

TABLE	PBIT	DPL	ACCESS	USE	BASE	LIMIT	SEGMENT NAME
GDT							
1	1	0	RW	16	FFFF0100H	00000037H	GDT:
2	1	0	RW	16	FFFF0000H	000000FFH	IDT:
3	1	0	EO	32	00000000H	FFFFFFFFH	_PHANTOM_CODE_
4	1	0	RW	32	00000000H	FFFFFFFFH	_PHANTOM_DATA_
LDT.1 (LDT1)							
1	1	0	RW	16	FFFF0138H	00000037H	LDT1:
2	1	0	ER	32	FFFF0270H	000000EDH	BITCOUNT.CODE32
3	1	0	RWD	32	00001260H	FFFFEFFFH	BITCOUNT.DATA
4	1	0	ER	32	FFFF0170H	0000005EH	BITCOUNT.INIT_CODE
5	1	0	RWD	32	00002260H	FFFFEFFFH	BITCOUNT.STACK
6	1	0	ER	32	FFFF0360H	00000051H	INTERRUPTS.INTRUPT_ROUTINES

80386 SYSTEM BUILDER
12/14/88 15:12:29   PAGE   4

GATE TABLE

GATE NAME	TABLE	PBIT	DPL	TYPE	WC	SELECTOR	OFFSET
INT1_GATE	IDT(1)	1	0	386INTR	0	GDT(3)	FFFF0364H
INT2_GATE	IDT(2)	1	0	386INTR	0	GDT(3)	FFFF0368H
INT3_GATE	IDT(3)	1	0	386INTR	0	GDT(3)	FFFF036CH
INT4_GATE	IDT(4)	1	0	386INTR	0	GDT(3)	FFFF0370H
INT5_GATE	IDT(5)	1	0	386INTR	0	GDT(3)	FFFF0374H
INT6_GATE	IDT(6)	1	0	386INTR	0	GDT(3)	FFFF0378H
INT7_GATE	IDT(7)	1	0	386INTR	0	GDT(3)	FFFF037CH
INT8_GATE	IDT(8)	1	0	386INTR	0	GDT(3)	FFFF0380H
INT9_GATE	IDT(9)	1	0	386INTR	0	GDT(3)	FFFF0384H
INT10_GATE	IDT(10)	1	0	386INTR	0	GDT(3)	FFFF0388H
INT11_GATE	IDT(11)	1	0	386INTR	0	GDT(3)	FFFF038CH
INT12_GATE	IDT(12)	1	0	386INTR	0	GDT(3)	FFFF0398H
INT13_GATE	IDT(13)	1	0	386INTR	0	GDT(3)	FFFF03A4H
INT14_GATE	IDT(14)	1	0	386INTR	0	GDT(3)	FFFF03A8H
INT16_GATE	IDT(16)	1	0	386INTR	0	GDT(3)	FFFF03ACH
INT0_GATE	IDT(0)	1	0	386INTR	0	GDT(3)	FFFF0360H

```
80386 SYSTEM BUILDER
12/14/88  15:12:29   PAGE    5

TASK TABLE

MAIN_TASK: TABLE    = GDT(6)      PBIT = 1  DPL = 0
           BASE     = FFFF0200H
           LIMIT    = 00000067H
           SS0:ESP0= GDT(4):00002260H
           SS1:ESP1= --------
           SS2:ESP2= --------
           SS:ESP   = GDT(4):00002260H
           PDR      = --------
           CS:EIP   = GDT(3):FFFF02B0H
           DS       = GDT(4)
           LDT      = --------
           IOPRIV   = 00H
           INTERRUPT NOT ENABLED
           DEBUG NOT ENABLED
           INITIAL

PROCESSING COMPLETED.      1 WARNING,      0 ERRORS
```

2.3 The Protected Flat Memory Model

You can easily create a system which combines the advantages of the flat memory model with the added value that the segmentation hardware can trap invalid data, code, and stack references. This variant of the flat memory model is called protected flat.

Remember that both the **_phantom_code_** and **_phantom_data_** segments must have the same base address. The default base address for these segments is 0. Use the **SEGMENT** definition in the build file to specify a different base address evenly divisible by 64K bytes.

The phantom segments do not have to span the entire address space, nor do they have to have the same limit. The default limit for these segments is 4 gigabytes - 1. Use the **SEGMENT** definition in the build file to specify different limits for one or both of the phantom segments. The minimum definition of a protected flat model system is one that has at least one of the phantom segments with a limit of less than 4 gigabytes - 1.

Figure 2-4 shows an example of a minimally protected flat memory model with **_phantom_data_**'s limit less than 4 gigabytes - 1. Reference to data outside of **_phantom_data_** causes a general protection fault.

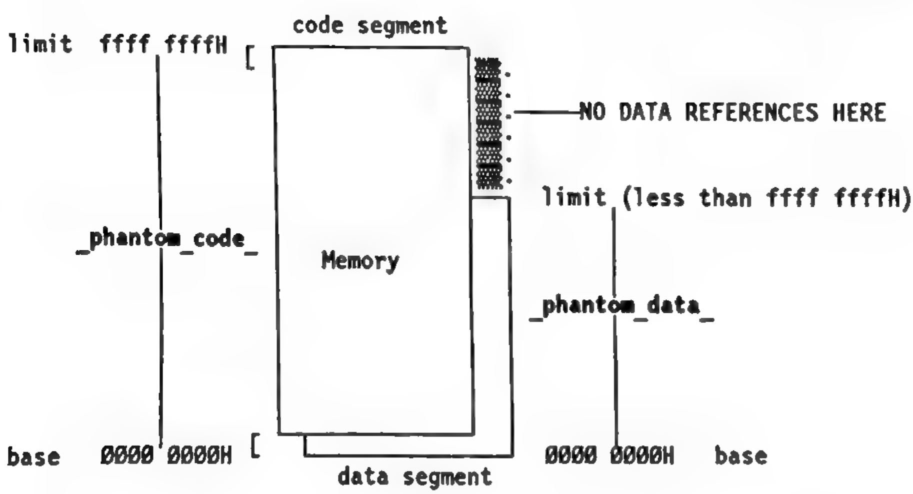

Figure 2-4 Minimally Protected Flat Memory Model

In the default flat model, the builder puts the stack with the data in **_phantom_data_**. Code and data are not protected from stack overflow. In the protected flat model, you can define a separate expand-down stack outside of **_phantom_data_**, adding the value of protecting the rest of the system from stack overflow. The expand-down stack has the same base address as the phantom segments. The builder sets its size to at least 4K bytes.

Use the build file **SEGMENT** definition to define the stack as **ED** (expand-down), and to define a limit less than 4 gigabytes - 1 but greater than 0 bytes. Do not specify a base address for the stack. Install the stack segment's descriptor in the GDT with the **TABLE** definition.

Figure 2-5 shows an example of a protected flat memory model with a separate expand-down stack. Although the phantom segments' limits are shown to be the same, they could be different as long as neither overlapped the expand-down stack. (Remember, memory wraps from high to low addresses on the Intel386 processors.)

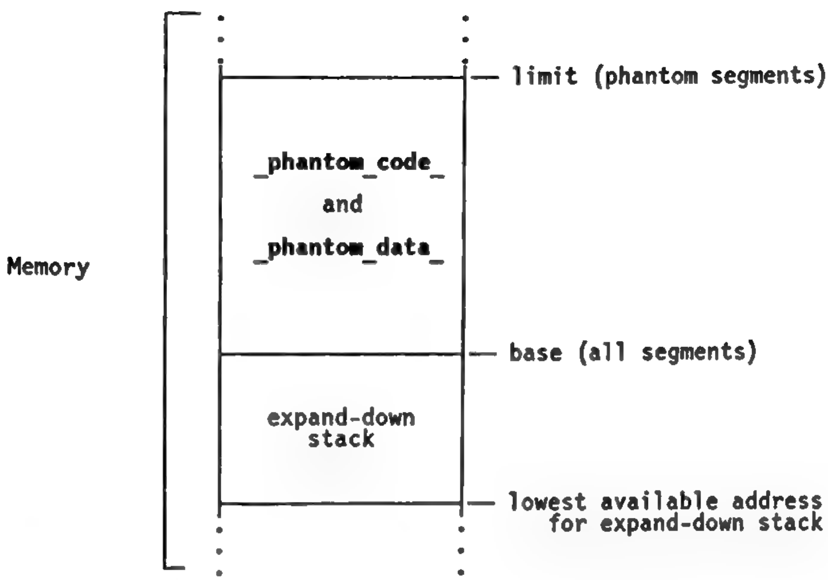

Figure 2-5 Protected Flat Memory Model with Expand-down Stack

2.4 The Protected Example Templates

The protected example templates differ from the embedded example templates in several ways. The phantom segments are not based at physical zero. The protected example demonstrates the creation of a separate expand-down stack. The application is written in assembly language. An interrupt procedure handles a dummy interrupt. Figure 2-6 illustrates the structure of the files that contain the source code for the protected example. See Figure 2-7 in Section 2.4.6 to see how the modules become a protected flat bootloadable system.

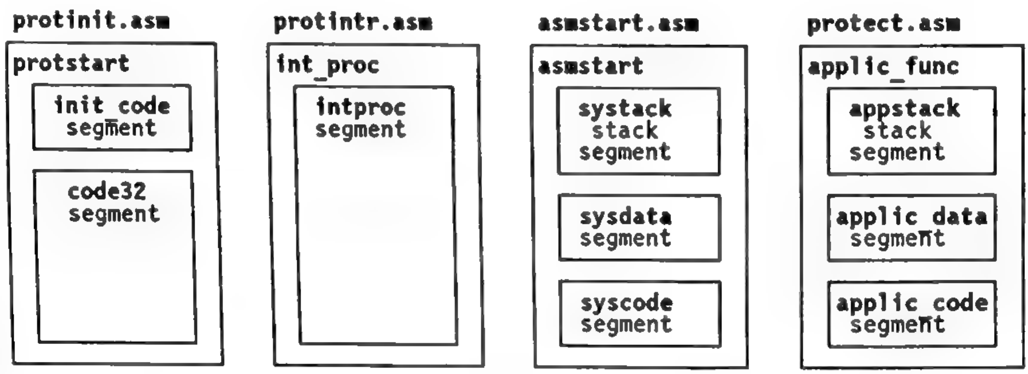

Figure 2-6 Protected Example Source Code

2.4.1 The Initialization Template

The initialization code template, **protinit.asm**, performs the same functions for the protected system as the initialization code for the embedded system. The first part, the **init_code** segment, places the microprocessor into 32-bit protected mode. The second part, **copytables** in the **code32** segment, copies the necessary parts of the descriptor tables created by the builder from ROM to RAM.

The major differences in code arise from the non-zero base of the phantom segments. The RAM addresses of the descriptor tables are still calculated by ANDing the corresponding ROM address with 0000ffffH. The **gdt_desc** and **idt_desc** still hold the absolute base

and limit values for the two tables. To copy a table and adjust its alias descriptor in the GDT, the code calculates the relative base for the table, which is the offset of the new table from the non-zero phantom base. (Remember, memory wraps from high to low addresses on the Intel386 processors.)

For simplicity, the code holds the values of the non-zero phantom base and the relative offset of absolute zero from this base. If the base of the phantom segments changes, the code must also change. (The code can be independent if it accesses the non-zero phantom base in one of the phantom segments' descriptors and calculates the relative offset of absolute zero instead.)

A listing of the **protinit.asm** file follows. Highlights point out the major added or changed code dealing with the non-zero base of the phantom segments.

```
; protinit.asm
; Initialization code for protected flat (linear) model example
;
; ****************************************************************************
;
; Version 2.0
; Copyright Intel Corp., 1988
; This template is intended for your benefit in developing applications/
; systems using Intel386(TM) family microprocessors.  Intel hereby grants
; you permission to modify and incorporate it as needed.
;
; ****************************************************************************
;
; This is initialization code to put the 386(TM) processor or the 376(TM)
; processor into protected flat mode.  It should work with all applications,
; but this model makes certain assumptions.  The memory model is a typical
; embedded application model:  descriptor tables and code reside in ROM
; and data is in RAM.  This example assumes that ROM begins at 0ffff0000H;
; since descriptor tables may need to be RAM-based for protected-mode
; execution, the code copies the builder-created GDT and IDT down into RAM
; with the RAM address being <ROM address AND 0ffffH> relative to the base
; of _phantom_code_ and _phantom_data_.  It assumes six GDT entries:  NULL,
; a GDT alias, an IDT alias, code, data, and a separate stack.  After
; entering protected mode, this code jumps to an ASM386 startup routine.
; You can change this JMP address to that of your code, or make the label
; of your code SYSTART.
;
; ****************************************************************************
;
;   Note:  Changing the base of the phantom segments in the build file
;          requires a similar change for the variables PHANTOM_BASE and
;          PHANTOM_ZERO.  The base of the phantom segments must be evenly
;          divisible by 64K bytes.
;
; ****************************************************************************
;
```

```
NAME protstart          ; name of object module

EXTRN systart:far       ; this is the label jumped to after init_code
                        ;  and copytables

pe_flag      equ 1      ; for setting PE bit
galias_off   equ 8      ; offset of GDT alias in GDT
ialias_off   equ 10H    ; offset of IDT alias in GDT
data_selc    equ 20H    ; offset of _phantom_data_ in GDT (GDT[4])
gdt_lim      equ 2fH    ; assume that 6 entries are all that are needed in GDT
idt_lim      equ 10fH   ; assume that 34 entries are all that are needed in IDT
phantom_zero equ 10000H     ; offset of absolute 0 from base of phantom segs
phantom_base equ 0ffff0000H ; base of phantom segments

CODEMACRO    opprefx        ; macro to change default operand size
             db 66H
ENDM

init_code    SEGMENT ER PUBLIC

;  GDT_DESC and IDT_DESC are public symbols referred to in the build file.
;  The LOCATION definitions in the TABLE section of the build file point to
;  these labels; the builder stores the base and limit for the named table
;  at this location in memory.

PUBLIC         gdt_desc
PUBLIC         idt_desc

gdt_desc       dp  ?
idt_desc       dp  ?

;  START is a label that points to the true beginning of our executable
;  code.  The BOOTSTRAP control causes the builder to place a short jump
;  to the named label (in this case, START) at the component reset vector.

PUBLIC         start

;  Since this code initializes either a 386 or 376 processor into protected
;  mode, the first instructions at START test for component type.
;  The 386 processor at reset is in real or compatibility mode:  the PE bit is
;  off and the D bit for CS is not set.  Instructions execute in their 16-bit
;  form.  The 376 processor at reset has the PE bit on as well as the D bit,
;  so instructions execute in their 32-bit form.

    nop                     ; NOPs are for initializing a 386
    nop                     ;  processor
start:
    cld                     ; clear direction flag
    smsw bx                 ; check for processor (376 or 386) at reset
    test bl,1               ; use SMSW rather than MOV for speed
    jnz pestart

;  Loading the GDTR at REALSTART or PESTART depends on user hardware
;  returning a READY after a write to ROM.

realstart:                  ; is a 386 processor and in 16-bit real mode
```

```
    opprefx                     ; use operand prefix to
    mov eax,offset gdt_desc     ; get 32-bit address of GDT pointer
    opprefx                     ; use operand prefix to
    and eax,Øffffh              ;  make address relative to reset area
    lgdtw cs:[eax]              ; load 24 bits of base into GDTR

    mov ax,bx                   ; copy machine status word
    or al,pe_flag               ; set PE bit
    lmsw ax                     ; load machine status word with PE bit set
    jmp next                    ; flush prefetch queue

pestart:                        ; is a 376 processor and in 32-bit protected mode
    mov eax,offset gdt_desc     ; get 32-bit address of GDT pointer
    and eax,Øffffh              ; make address relative to reset area
    lgdt cs:[eax]               ; load 32 bits of base into GDTR

next:
    xor eax,eax                 ; initialize data selectors
    mov al,data_selc            ; GDT[4] is _phantom_data_
    mov ds,ax
    mov ss,ax
    mov es,ax
    mov fs,ax
    mov gs,ax
    test bl,1
    jnz pejump

;  Use SYSTART as the destination of this next jump if your hardware does
;  return a READY on a write to ROM, and skip the COPYTABLES routine.

    opprefx                     ; use operand prefix for 386 processor jump
pejump:
    jmp far ptr copytables      ; first far jump causes A31-2Ø to drop low

init_code ENDS

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

code32  SEGMENT ER PUBLIC

copytables:

;  Copy GDT and IDT from ROM to RAM.  Assume the RAM address for the tables
;  is the absolute ROM address AND ØØØØffffH.  This code uses the
;  second GDT entry, the GDT alias at GDT[1] = (base of GDT) + galias_off.

    mov eax,offset gdt_desc     ; get address of gdt_desc
    mov ebx,dword ptr [eax]+2   ; base of GDT is 2 bytes past gdt_desc
                                ; note that ebx holds absolute base of GDT

;  Move the GDT descriptors from ROM to RAM.

    mov esi,ebx                 ; source of move is base of GDT relative to
    xor esi,phantom_base        ;  phantom segments
    and ebx,ØffffH              ; displacement of GDT in ROM
    add ebx,phantom_zero        ; calculate offset of new GDT
    mov edi,ebx                 ; destination of move is calculated address
    mov ecx,gdt_lim+1           ; count of move is 6 entries X 8 bytes each
```

```
    rep movsb                    ; move 6 descriptors from ROM to RAM

;  Modify the GDT alias at GDT[1] to reflect the new base and limit of the
;  RAM-based GDT.  The GDT alias is a data segment descriptor (read/write)
;  which allows future modification of the GDT.  Reload the GDTR with the
;  new base and limit values.  We do this by changing the GDT alias
;  to reflect the new base and limit, saving the changes, setting up the GDT
;  alias to reload the GDTR, reloading, and then restoring the GDT alias.
;  EBX holds the new base address of the GDT (relative).  Note that the GDT
;  alias holds the absolute base of the GDT (not relative to the
;  phantom segments' base).

                                 ; change base in second dword of GDT alias
    and dword ptr [ebx]+galias_off+4,0ffff00H
                                 ; change limit in GDT alias
    mov word ptr [ebx]+galias_off,gdt_lim
                                 ; save part of GDT alias
    mov edx,dword ptr [ebx]+galias_off+2
                                 ; get old absolute base for GDT
    mov ecx,dword ptr [eax]+2
                                 ; calculate new absolute base for GDT
    and ecx,0ffffH
                                 ; set up new base for loading GDTR
    mov dword ptr [ebx]+galias_off+2,ecx
                                 ; reload the GDTR
    lgdt pword ptr [ebx]+galias_off
                                 ; restore saved part of GDT alias
    mov dword ptr [ebx]+galias_off+2,edx

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  IDT mov from ROM to RAM

    mov eax,offset idt_desc    ; get address of idt_desc
    mov edx,dword ptr [eax]+2  ; base of IDT is 2 bytes past idt_desc
                               ; note that edx holds absolute base of IDT

;  Move the IDT descriptors from ROM to RAM.

    mov esi,edx                ; source of move is base of IDT relative to
    xor esi,phantom_base       ;  phantom segments
    and edx,0ffffH             ; displacement of IDT in ROM
    add edx,phantom_zero       ; calculate offset of new IDT
    mov edi,edx                ; destination of move is calculated address
    mov ecx,idt_lim+1          ; count of move is 34 entries X 8 bytes each
    rep movsb                  ; move 34 descriptors from ROM to RAM

;  Modify the IDT alias descriptor at GDT[2] to reflect the new base and limit
;  values for the RAM-based IDT and load the IDTR.  EAX holds the address of
;  IDT_DESC.  EBX holds the address of the new GDT (relative).  Note that the
;  IDT alias holds the absolute base of the IDT (not relative to the phantom
;  segments' base).

                               ; change base in second dword of IDT alias
    and dword ptr [ebx]+ialias_off+4,0ffff00H
                               ; change limit in IDT alias
    mov word ptr [ebx]+ialias_off,idt_lim
                               ; save part of IDT alias
    mov edx,dword ptr [ebx]+ialias_off+2
```

```
                                ; get old absolute base of IDT
    mov ecx,dword ptr [eax]+2
                                ; calculate new absolute base of IDT
    and ecx,0ffffH
                                ; set up base for loading IDTR
    mov dword ptr [ebx]+ialias_off+2,ecx
                                ; load the IDTR
    lidt pword ptr [ebx]+ialias_off
                                ; restore saved part of IDT alias
    mov dword ptr [ebx]+ialias_off+2,edx

    jmp systart                 ; jump to startup code

code32 ENDS

END
```

2.4.2 The Build File Template

The **protect.bld** build file containing system definitions is where the work is done to define the protected flat memory model with an expand-down stack. The following explains the major changes.

SEGMENT definition — describes the segments. All of the segments get a descriptor privilege level of zero, or most privileged. One input segment becomes the separate expand-down stack. The phantom segments' base is set to a non-zero value evenly divisible by 64K bytes, and their limits are set to values less than 4 gigabytes - 1. The **_phantom_code_** segment's limit is less than the **_phantom_data_** segment's limit.

TABLE definition — (first occurrence) defines the global descriptor table and explicitly places the expand-down stack descriptor in the sixth slot, GDT[5].

TABLE definition — (second occurrence) defines the interrupt descriptor table and explicitly places the interrupt gate in the 34th slot, IDT[33].

A listing of the **protect.bld** build file follows. Highlights point out the major changes.

```
-- protect.bld
-- Build file for input to BLD386 to create protected flat model example
--
-- ***************************************************************************
--
-- Version 2.0
-- Copyright Intel Corp., 1988
-- This template is intended for your benefit in developing applications/
-- systems using Intel386(TM) family microprocessors.  Intel hereby
-- grants you permission to modify and incorporate it as needed.
--
-- ***************************************************************************
--

protflat; -- build program id

SEGMENT
    asmstart                        -- Give all user segments a DPL of 0.
        (DPL = 0),                  -- The name "asmstart" is the module-id
                                    -- for all segments because its object
                                    -- module is first in the BND386 input
                                    -- list.
    appstack                        -- Protected stack is expand-down.
        (ED,                        -- A limit for the stack must be specified
         LIMIT = 050H),             -- greater than 0 and less than 4 gigabytes,
                                    -- but the stack size is 4K bytes
                                    -- (or larger in 4K-byte steps).

                                    -- The two _phantom_ segments are created
                                    -- by the builder when the FLAT control
    _phantom_code                   -- is used.
        (BASE = 0ffff0000H,         -- Base values of both _phantom_ segments
         LIMIT = 0ff00H),           -- must be the same, and evenly divisible
    _phantom_data                   -- by 64K bytes.
        (BASE = 0ffff0000H,
         LIMIT = 020000H);          -- Limit values of _phantom_ segments can
                                    -- differ.

 TASK                               -- Task is for ICE(TM)-386 or
    main_task                       -- ICE(TM)-376 emulator initialization.
        (BASE = 0ffff0300H,
         DATA = sysdata,            -- Points to a segment that
                                    -- indicates initial DS value.
         CODE = systart,            -- Entry point is systart, which
                                    -- must be a public id.
         STACKS = (systack),        -- Segment id points to stack
                                    -- segment. Sets the initial SS:ESP.
         NO INTENABLED              -- Disable interrupts.
        );

TABLE        -- create GDT
   GDT                              -- GDT_DESC is a public symbol in
                                    -- the "protstart" initialization module.
       (LOCATION = gdt_desc,        -- In a buffer starting at GDT_DESC,
                                    -- BLD386 places the GDT base and
                                    -- GDT limit values. Buffer must be
                                    -- 6 bytes long.  The base and limit
                                    -- values are placed in this buffer
```

```
                                        -- as two bytes of limit plus
                                        -- four bytes of base in the format
                                        -- required for use by LGDT instruction.
        BASE = 0ffff0200H,
        ENTRY = (5:appstack)            -- Explicitly install separate expand-down
                                        -- stack in GDT[5].
       );  -- end GDT

GATE                                    -- Define an interrupt gate for the
    int_gate                            -- interrupt handler.
        (DPL = 0,
        ENTRY = intproc,
        INTERRUPT);

TABLE       -- create IDT
    IDT                                 -- IDT_DESC is a public symbol in the
       (LOCATION = idt_desc,            -- "protstart" initialization module.
                                        -- In the buffer starting at IDT_DESC
                                        -- the builder places two bytes of the
                                        -- IDT limit and four bytes of the IDT
                                        -- base values in the format required
        BASE = 0ffff0000H,              -- for use by LIDT instruction.
                                        -- Slots 0 through 31 are Intel-reserved.
        ENTRY = (33:int_gate)
       );  -- end IDT

MEMORY
    (RESERVE = (0..300H),               -- For copying down IDT and GDT.
     RANGE = (        -- begin configuration section --
           rom_area = ROM(0ffff0000H..0fffffff0H),
           ram_area = RAM(400H..0ffffH)
          )           -- end configuration section   --
    );

TABLE
    ldtl (NOT CREATED);                 -- Builder does not place LDT in object
                                        -- module, but contents appear in listing.

END
```

2.4.3 The Interrupt Routine Template

One interrupt procedure is defined for a non-reserved interrupt. A listing of the **protintr.asm** file follows.

```
; protintr.asm
; ASM386 interrupt stub
;
; ****************************************************************************
;
;  Version 2.0
;  Copyright Intel Corp., 1988
;  This template is intended for your benefit in developing applications/
;  systems using Intel386(TM) family microprocessors.  Intel hereby
;  grants you permission to modify and incorporate it as needed.
;
; ****************************************************************************
;
NAME int_proc

PUBLIC intproc

intcode SEGMENT EO PUBLIC

intproc PROC FAR

int33:
    hlt
    iretd

intproc ENDP

intcode ENDS

END
```

2.4.4 The Assembler Application Startup Template

The code in the startup template for an assembler application defines a system stack, initializes the stack pointer, calls the application, and generates an interrupt. A listing of the **asmstart.asm** file follows.

```
; asmstart.asm
; ASM386 system startup code for protected flat model example
;
; ****************************************************************************
;
; Version 2.0
; Copyright Intel Corp., 1988
; This template is intended for your benefit in developing applications/
; systems using Intel386(TM) family microprocessors.  Intel hereby
; grants you permission to modify and incorporate it as needed.
;
; ****************************************************************************
;
NAME    asmstart

PUBLIC  systart

EXTRN   app_entry:FAR

systack STACKSEG 100

sysdata SEGMENT RW
var2    dd 5
sysdata ENDS

syscode SEGMENT EO PUBLIC

ASSUME  DS:sysdata, SS:systack

systart:
  inc esi                        ; used as marker for tracing execution
  mov esp, stackstart systack    ; initialize esp
  call app_entry                 ; go to applic code
  inc esi                        ; marker
  int 33                         ; cause an interrupt
  inc esi                        ; marker

syscode ENDS

END systart, DS:sysdata, SS:systack
```

A listing of the **protect.asm** application follows.

```
; protect.asm
; ASM386 application code to demonstrate protected flat model
;
; ****************************************************************************
;
; Version 2.0
; Copyright Intel Corp., 1988
; This template is intended for your benefit in developing applications/
; systems using Intel386(TM) family microprocessors.  Intel hereby
; grants you permission to modify and incorporate it as needed.
;
; ****************************************************************************

NAME     applic_func

PUBLIC   app_entry, error_entry, var1

appstack STACKSEG 40            ; This is the expand-down stack

applic_data SEGMENT RW PUBLIC
var1     dd ?
applic_data ENDS

ASSUME   DS:applic_data, SS:appstack

applic_code SEGMENT ER PUBLIC

const    dd 5

app_entry:
    mov ebp,esp
    mov ax ,appstack             ; get selector for appstack
    mov ss ,ax                   ; set SS selector
    mov esp,stackstart appstack  ; get initial ESP value
    mov ebx,offset const         ; get CS relative offset
    mov eax,[ebx]                ; see if we get the constant from code
    push eax                     ; test stack access
    push eax                     ; test stack access
    pop var1                     ; test store
    sub esp,4088                 ; reduce ESP for lesser loops

stack_loop:
    push eax                     ; loop until stack limit exception
    cmp eax,esp                  ; insert your own limit comparison
    jne error_entry
    inc eax
    jmp stack_loop
;
;   This does nothing except prove that the system works.
;
error_entry:
    mov ax, ds                   ; restore SS with system stack selector
    mov ss, ax
    mov esp,ebp                  ; restore ESP
    ret                          ; go back and cause interrupt

applic_code ENDS
END
```

2.4.5 Creating the Protected Flat Example System

You create the protected example system with essentially the same commands as the embedded example system. The binder links the interrupt module with the other modules to resolve symbolic addressing. The commands for assembling, binding, and building follow.

VMS:	DOS:
`asm386/debug protinit.asm`	`asm386 protinit.asm debug`
`asm386/debug protect.asm`	`asm386 protect.asm debug`
`asm386/debug asmstart.asm`	`asm386 asmstart.asm debug`
`asm386/debug protintr.asm`	`asm386 protintr.asm debug`

VMS:

```
bnd386/noload/debug/object=protect.bnd asmstart.obj, protect.obj,
   protinit.obj, protintr.obj
```

DOS:

```
bnd386 asmstart.obj,protect.obj,protinit.obj,protintr.obj noload debug
   object (protect.bnd)
```

VMS:

```
bld386/buildfile=protect.bld/bootstrap=start/flat/mod376 protect.bnd
```

DOS:

```
bld386 protect.bnd buildfile(protect.bld) bootstrap(start) flat mod376
```

2.4.6 The Protected Flat Example System

Figure 2-7 shows what the memory looks like with the location of code, data, and system data structures after building the protected example system.

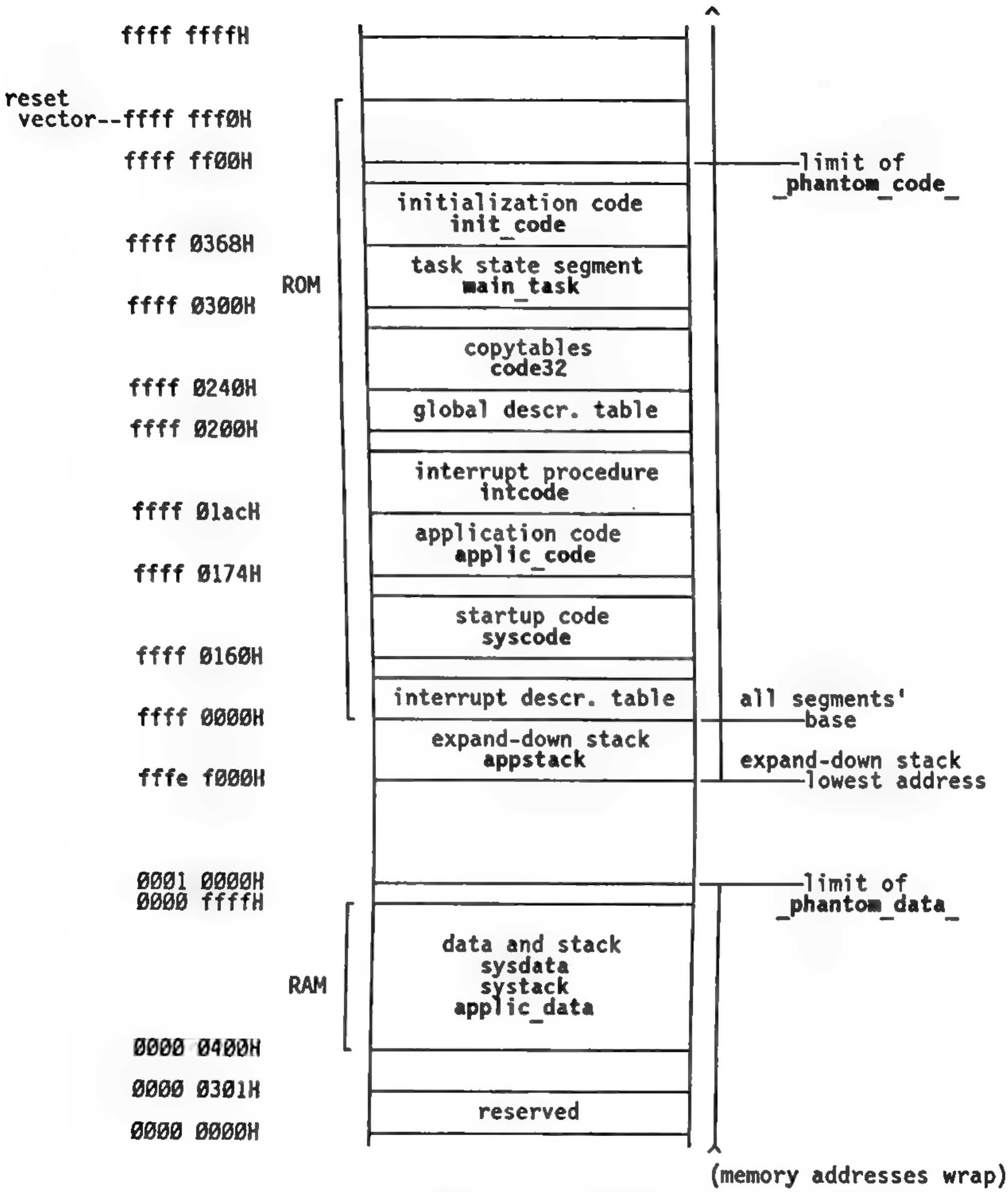

Figure 2-7 Protected Example System Memory Map

The maps portion of the builder listing file for the **protect** example system follows.

```
80386 SYSTEM BUILDER
12/14/88  13:33:02   PAGE    4


SEGMENT MAP

TABLE     PBIT DPL ACCESS USE  BASE      LIMIT     SEGMENT NAME

GDT

 1          1    0    RW     16 FFFF0200H  0000003FH  GDT:
 2          1    0    RW     16 FFFF0000H  0000010FH  IDT:
 3          1    0    EO     32 FFFF0000H  0000FF00H  _PHANTOM_CODE_
 4          1    0    RW     32 FFFF0000H  00020000H  _PHANTOM_DATA_
 5          1    0    RWD    32 FFFF0000H  FFFFEFFFH  ASMSTART.APPSTACK

LDT.1 (LDT1)

 1          1    0    RW     16 FFFF0110H  0000004FH  LDT1:
 2          1    0    EO     32 FFFF0160H  00000010H  ASMSTART.SYSCODE
 3          1    0    RW     32 00000400H  00000006H  ASMSTART.SYSDATA
 4          1    0    RWD    32 00001408H  FFFFEFFFH  ASMSTART.SYSTACK
 5          1    0    ER     32 FFFF0174H  00000037H  ASMSTART.APPLIC_CODE
 6          1    0    RW     32 00001408H  00000006H  ASMSTART.APPLIC_DATA
 7          1    0    ER     32 FFFF0240H  00000099H  ASMSTART.CODE32
 8          1    0    ER     32 FFFF0368H  0000005EH  ASMSTART.INIT_CODE
 9          1    0    EO     32 FFFF01ACH  00000001H  ASMSTART.INTCODE


80386 SYSTEM BUILDER
12/14/88  13:33:02   PAGE    5


GATE TABLE

GATE NAME   TABLE          PBIT DPL  TYPE     WC  SELECTOR       OFFSET

INT_GATE    IDT(33)          1    0  386INTR   0  GDT(3)         000001ACH
```

```
TASK TABLE

MAIN_TASK: TABLE    = GDT(7)      PBIT = 1  DPL = 0
           BASE     = FFFF0300H
           LIMIT    = 00000067H
           SS0:ESP0= GDT(4):00001408H
           SS1:ESP1= --------
           SS2:ESP2= --------
           SS:ESP   = GDT(4):00001408H
           PDR      = --------
           CS:EIP   = GDT(3):00000160H
           DS       = GDT(4)
           LDT      = --------
           IOPRIV   = 00H
           INTERRUPT NOT ENABLED
           DEBUG NOT ENABLED
           INITIAL

PROCESSING COMPLETED.      2 WARNINGS,      0 ERRORS
```

TEMPLATE FILE NAMES FOR V1.1 AND V2.Ø	
V2.Ø File name	V1.1 File name(s)
flatinit.asm	flat.a38
simpinit.asm	flatsim.a38
protinit.asm	protflat.asm (DOS) protflat.a38 (VMS)
flatintr.asm	intrpt.a38
protintr.asm	intproc.asm
cstart.asm	cstart.a38
asmstart.asm	sys.asm
bitcount.c	bitcnt.c
reverse.c	revrs.c
simple.c	simple.c
protect.asm	applic.asm
flat.bld	flat.bld
simple.bld	simple.bld
protect.bld	applic.bld
flat.bat (DOS) flat.com (VMS)	flat.bat (DOS) flat.com (VMS)
simple.bat (DOS) simple.com (VMS)	flatsim.bat (DOS) flatsim.com (VMS)
protect.bat (DOS) protect.com (VMS)	prot.bat (DOS) prot.com (VMS)
showbitc.inc	showbitc.inc

TEMPLATE FILE NAMES FOR V1.1 AND V2.Ø (continued)	
V2.Ø File name	V1.1 File name(s)
showrevr.inc	showrevr.inc
showsimp.inc	simple.inc
showprot.inc	sys.inc and sys1.inc

Italicized words and phrases in a definition refer to other glossary entries.

16-bit real mode	the mode of execution on the 386™ microprocessor where the processor executes instructions in their 16-bit form. The *protection-enable (PE) bit* is off in the *machine status word*. Address calculation mimics the 8086 processor. The 386 microprocessor executes in this mode on reset.
32-bit protected mode	the mode of execution on the Intel386™ family microprocessors where the processor executes instructions in their 32-bit form. The *protection-enable (PE) bit* is on in the *machine status word*. The first *far jump* has been executed. This mode uses *selectors* and *descriptors* to calculate addresses. The 376™ microprocessor executes only in this mode. See also *protection*.
32-bit segment	a *segment* whose use-attribute is USE32. This attribute is the default for a segment. The use-attribute is represented by bit 54 (B/D) in a segment's *descriptor*. By default, addresses and operands are 32 bits long. A 32-bit segment naturally corresponds to executing in *32-bit protected mode*, but *operand prefixes* preceding some instructions allow the processor to execute a 32-bit segment in *16-bit real mode*. See also *D bit*.
absolute addresses	fixed locations in memory for data and code (contrary to relocatable addresses, which could change from execution to execution). See also *bootloadable system*, *physical address*, and *relocating*.
access attributes	characteristics which define the type of a *segment*: read-only data, read-write data, execute-read code, or execute-only code. These attributes are represented by bits 41 (W/R) and 43 (Executable) in a segment's *descriptor*.

ACCESS bit — bit 24 in a segment's *descriptor* that is set when the segment *selector* has been loaded into a *segment* register or used by selector test instructions. The ACCESS bit in the *global descriptor table*'s *alias descriptor* is set on execution of a load *global descriptor table register* (LGDT) instruction.

alias, alias descriptor — a segment *descriptor* which has *access attributes* contrary to the normal use of the *segment*. The *system builder* always creates a *global descriptor table* with an alias descriptor for the global descriptor table itself in GDT[1] and an alias descriptor for the *interrupt descriptor table* in GDT[2]. These aliases have access attributes that describe the *tables* as writable data segments.

base, base address — the lowest *offset* of the item being discussed (except *expand-down* segments). Typical items are *segments*, *tables*, and *task state segments*. In a flat memory model, if the phantom segments have a base address of zero, all base addresses are the same as their *physical addresses*. See also *relative base* and *expand-down*.

binder, BND386 — the RLL (DOS) or R&L (VMS) utility that performs linking. The binder combines *segments* with like names and resolves symbolic addressing. The binder can produce a loadable *module* (relocatable) or linkable *module* (suitable for input to the binder or the *system builder*). The binder cannot produce a *bootloadable system*.

bootloadable system — an object *module* with *absolute addresses*. A simple bootstrap loader can load a bootloadable system onto a target processor. The system contains code which initializes the processor on reset. See also *system builder*.

bootstrap jump — the instruction located at the *reset vector*. Usually this instruction is a *near jump* to the initialization code.

Italicized words and phrases in a definition refer to other glossary entries.

builder, BLD386	See *system builder.*
build file	a file of system implementation definitions used by the *system builder* to create a *bootloadable system.* The definitions describe *system data structures*, initial values for the system, and *memory configuration.*
compatibility mode	See *16-bit real mode.*
component	microprocessor.
D bit	bit 54 (B/D) in a segment's *descriptor.* The D bit refers to the default operand size of a code *segment.* If the bit is 1, the default operand size is 32 bits. If the bit is 0, the default operand size is 16 bits. See also *32-bit segment.*
descriptor	(1) eight bytes of memory containing the *base*, *limit*, and *access attributes* for a given region of linear address space such as a *segment*, *table*, or *task state segment.* Usually "descriptor" alone refers to the descriptor for a code or data segment. "*Alias descriptor*" most often refers to the descriptor for a table. "Task descriptor" or "TSS descriptor" refers to the descriptor for a task state segment. (2) a *gate.*
descriptor privilege level (DPL)	bits 29 and 30 in a segment's *descriptor.* The *segmentation hardware* checks descriptor privilege levels on accesses to code and data *segments* to ensure that the referring code has sufficient privilege. A flat model system has all segments at privilege level 0, which obviates the privilege-level checking. See also *privilege levels*, *privileged instructions*, and *protection.*

direction flag	bit 10 in the EFLAGS register. The value of this bit defines whether ESI and/or EDI registers postdecrement or postincrement during string instructions.
entry point	the first instruction to be executed in a code *segment* or *task*.
expand-down	a special kind of data *segment* useful for stacks. Stack growth is down from the *base address*. The expand-down attribute is in bit 42 of the segment's *descriptor*. A software system can dynamically increase the expand-down segment's size by lowering the *limit* in the segment's descriptor. A 32-bit expand-down segment's base address is one byte higher than its highest available address. Its lowest available address is the sum of its base plus limit plus one, ignoring any carry past 32 bits of address. Its size is the complement of its limit, or the difference of its base and lowest available address. Its minimum size is 4K bytes.
far jump	an intersegment jump; this direct control-transfer instruction has a *selector* and *offset* representing the destination, which allows control to transfer to (possibly) a different code *segment*.
fault handler	code which executes when a fault, or interrupt, is raised.
gate	eight bytes of memory used to regulate transfer of control to another code *segment*. A gate is sometimes called a *descriptor* because it has a layout similar to a segment descriptor. It contains information about the *selector* for the target segment, the *offset* of the *entry point* within the target segment, the type and *privilege level* of the gate, and the number of words to pass on the stack. Gates provide indirection that allows the processor to perform *protection* checks. Gate types can be call, interrupt, trap, or *task*.

Italicized words and phrases in a definition refer to other glossary entries.

general protection fault	interrupt number 13 raised on a *protection* violation such as exceeding a segment *limit*, violating the *access attributes* of a *segment*, or violating *privilege levels*.
global descriptor table (GDT)	an array of *descriptors* defining *segments* and *gates* available for use by all *tasks* in the system. There is only one global descriptor table for a software system.
global descriptor table register (GDTR)	the system register that contains the *base* and *limit* of the *global descriptor table*.
interrupt descriptor table (IDT)	an array of *task*, interrupt, and/or trap *gates* that act as interrupt vectors; each gate's *slot* number is the interrupt number. There is only one interrupt descriptor table for a software system.
interrupt descriptor table register (IDTR)	the system register that contains the *base* and *limit* of the *interrupt descriptor table*.
interrupt stubs	*routines* that execute when an interrupt occurs but do essentially nothing.
limit	the *offset* of the farthest available byte from the *base address* in a *segment* that is not *expand-down*. This value is one less than the number of bytes in the segment. For a 32-bit expand-down segment, the limit is the complement of the number of bytes in the segment.
linear addresses	the same as *physical addresses* in a memory model which does not use any virtual memory.
linker	See *binder*.

local descriptor table (LDT) — an array of *descriptors* defining *segments* and *gates* protected from use by all but specified *tasks* in the system. Tasks that have a pointer to a local descriptor table in their *task state segment* can access that table. Sometimes the *global descriptor table* also holds descriptors for local descriptor tables. There can be many local descriptor tables in a software system.

local descriptor table register (LDTR) — the system register that contains the *selector* for the *descriptor* of the currently active *local descriptor table.*

machine status word — bits 0 through 15 of control register 0 (CR0); contains the *protection-enable (PE) bit.*

module — (1) a file of code in some stage of translation. An object module refers to the output of an assembler, compiler, *binder*, or *system builder*. An input module refers to a file in the form accepted by translating, linking, or building software. (2) a unit of code with some purpose, not necessarily an entire file, and possibly spanning several files.

near jump — an intrasegment jump; a direct control-transfer instruction with an *offset* representing the destination, thus allowing control to transfer to code within the same code *segment.*

offset — the displacement; the number of units (usually bytes) away from the zeroth location in memory, or the number of units away from the *base address* of the enclosing *segment* or data structure.

operand prefix — a byte of memory with value 66H preceding an instruction. The processor uses an operand size other than the default when executing this instruction.

overlaid segments — *segments* whose *base* and *limit* values cause them to share some or all of their *physical addresses*.

Italicized words and phrases in a definition refer to other glossary entries.

physical address	the *offset* of the first byte of an item from the zeroth byte in memory. The minimum physical address on the Intel386 microprocessors is zero; the maximum physical address is 4 gigabytes - 1, or ffffffffH. See also *absolute addresses.*
prefetch queue	the instruction pipeline on the Intel386 microprocessors.
privilege level	one of four values: 0 (most privileged), 1, 2, or 3 (least privileged). The *descriptor privilege level (DPL)* of the currently executing code *segment* is also called the current privilege level (CPL). See also *selector.*
privileged instructions	instructions that affect system registers or halt the processor. These instructions can only be executed when the current *privilege level* is 0.
protection	the mechanisms implemented by the hardware of the Intel386 microprocessors, especially when the *protection-enable (PE) bit* is on and the first *far jump* has been executed. There are five basic kinds of protection available: type checking, *limit* checking, restriction of addressable domain, restriction of *entry points*, and restriction of instruction set. See also *general protection fault*, *privileged instructions*, and *segmentation hardware.*
protection-enable (PE) bit	bit 0 in the *machine status word.* If PE is 1, the processor executes in *32-bit protected mode.* If PE is 0, the processor operates in *16-bit real mode.*
protection level	See *descriptor privilege level (DPL)*, and *privilege level.*
real mode	See *16-bit real mode.*

relative base, relative offset	the *offset* of the lowest address of the item being discussed from some other known address. An item such as a *table* that is located within a larger *segment* has a *base address* relative to the base address of the larger segment. If the larger segment has a physical base address of zero, the table's base and relative base are the same *physical address*.
relocating	the process of changing the *physical address* of an item, and adjusting all references to that item.
reset vector	*physical address* fffffff0H on the Intel386 famly processors. The processor executes the instruction at this location first on reset. See also *bootstrap jump*.
routine	a piece of executable code with a well-defined *entry point* and exit point. A routine is similar to a subroutine but may be jumped to or executed on a trap or interrupt rather than called.
segment	a continuous piece of memory defined by a *base address* and a *limit*. See also *descriptor*, *32-bit segment*, *expand-down*, and *overlaid segments*.
segmentation hardware	the parts of the Intel386 family microprocessors that implement *protection* for accessing *segments*. See also *general protection fault*.
selector	a *system data structure* used in computing an address that identifies a *descriptor* by specifying a descriptor *table* and indexing a descriptor within that table. A selector also contains a requested *privilege level* (RPL), which is the *descriptor privilege level (DPL)* of the referring *segment*. Currently available *segments* are represented by selectors in the segment registers CS, DS, SS, ES, FS, and GS.
slot	a particular 8-byte position in a descriptor *table*.

Italicized words and phrases in a definition refer to other glossary entries.

system builder, BLD386	the RLL (DOS) or R&L (VMS) utility that creates a *bootloadable system* from linkable input *modules* and system definitions in a *build file*.
system data structures	regions of continuous *linear address* space with values in defined positions. *Descriptors*, *tables*, *gates*, *selectors*, and *task state segments* are the most common system data structures. Sometimes discussion of system data structures includes the system registers.
table	an array of *descriptors* and/or *gates*. Each active table has a system register that points to the table. See also *global descriptor table*, *interrupt descriptor table*, and *local descriptor table*.
task	(1) the code, data, and *system data structures* which collectively define a sequential thread of execution. (2) one or more of the system data structures associated with a task: *task state segments*, task or TSS *descriptor*, or task *gate*.
task state segment (TSS)	a *system data structure* of a minimum of 68H byt that at least describes the processor state (such as contents of registers) upon entry to or exit from a *task*. Other literature refers to this type of data structure as a task control block.

F

G

H-I

P

R

S

International Sales Offices

BELGIUM
Intel Corporation SA
Rue des Cottages 65
B-1180 Brussels

DENMARK
Intel Denmark A/S
Glentevej 61-3rd Floor
dk-2400 Copenhagen

ENGLAND
Intel Corporation (U.K.) LTD.
Piper's Way
Swindon, Wiltshire SN3 1RJ

FINLAND
Intel Finland OY
Ruosilante 2
00390 Helsinki

FRANCE
Intel Paris
1 Rue Edison-BP 303
78054 St.-Quentin-en-Yvelines Cedex

ISRAEL
Intel Semiconductors LTD.
Atidim Industrial Park
Neve Sharet
P.O. Box 43202
Tel-Aviv 61430

ITALY
Intel Corporation S.P.A.
Milandfiori, Palazzo E/4
20090 Assago (Milano)

JAPAN
Intel Japan K.K.
5-6 Tokodai, Tsukuba-shi
Ibaraki, 300-26

NETHERLANDS
Intel Semiconductor (Nederland B.V.)
Alexanderpoort Building
Marten Meesweg 93
3068 Rotterdam

NORWAY
Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013, Skjetten

SPAIN
Intel Iberia
Calle Zurbaran 28-IZQDA
28010 Madrid

SWEDEN
Intel Sweden A.B.
Dalvaegen 24
S-171 36 Solna

SWITZERLAND
Intel Semiconductor A.G.
Talackerstrasse 17
8125 Glattbrugg
CH-8065 Zurich

WEST GERMANY
Intel Semiconductor GmbH
Seidlestrasse 27
D-8000 Muenchen 2

intel®

Program Development Templates 481894-001

READER RESPONSE CARD

We'd Like Your Opinion

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future versions.

To order publications, contact the Intel Literature Department (see page ii of this manual).

Fill in the squares below with a rating of 1 through 10:

POOR				AVERAGE					EXCELLENT
1	2	3	4	5	6	7	8	9	10

- ☐ Readability
- ☐ Technical depth
- ☐ Technical accuracy
- ☐ Usefulness of material for your needs
- ☐ Comprehensibility of material
- ☐ OVERALL QUALITY OF THIS MANUAL

If you gave a 4 or less (in any category), please explain here:

____

____

____

What suggestions would you have for improving this manual:

____

____

If you would like us to call you for more specific suggestions about this book, please additionally fill in your phone number below.

Name ____________________________________

Phone Number (________)____________________________

Address ____________________________________

____

____

Thanks for taking the time to fill out this form.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 79 HILLSBORO, OR

POSTAGE WILL BE PAID BY ADDRESSEE

INTEL CORPORATION
DTO TECHNICAL PUBLICATIONS HF2-38
5200 NE ELAM YOUNG PARKWAY
HILLSBORO OR 97124-9978

Please fold here and close the card with tape. Do not staple.

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the other side of this form will help us produce better manuals. Each reply will be reviewed. All comments and suggestions become the property of Intel Corporation.

If you are in the United States and are sending only this card, postage is prepaid.

If you are sending additional material or if you are outside the United States, please insert this card and any enclosures in an envelope. Send the envelope to the above address, adding "United States of America" if you are outside the United States.

Thanks for your comments.

INTEL CORPORATION
5200 N.E. Elam Young Pkwy.
Hillsboro, Oregon 97123